

The Python Tutorial

Learning Notes

Du Ang
du2ang233@gmail.com

September 13, 2017

Contents

1	Whetting Your Appetite	4
1.1	Some applications of Python	4
1.2	The features of Python	4
1.3	The origin of the name "Python"	4
2	Using the Python Interpreter	4
2.1	Invoking the Interpreter	4
2.1.1	Argument Passing	4
2.1.2	Interactive Mode	5
2.2	The Interpreter and Its Environment	5
2.2.1	Source Code Encoding	5
3	An Informal Introduction to Python	5
3.1	Using Python as a Calculator	5
3.1.1	Numbers	5
3.1.2	Strings	6
3.1.3	Unicode Strings	10
3.1.4	Lists	10
3.2	First Steps Towards Programming	11
4	More Control Flow Tools	12
4.1	if Statements	12
4.2	for Statements	12
4.3	The range() function	13
4.4	break and continue Statements, and else Clauses on Loops	13
4.5	pass Statements	14
4.6	Defining Functions	14
4.7	More on Defining Functions	16
4.7.1	Default Arguments Values	16
4.7.2	Keyword Arguments	17
4.7.3	Unpacking Arugument Lists	18
4.7.4	Lambda Expressions	18
4.7.5	Documentation Strings	19
4.8	Intermezzo: Coding Style	19
5	Data Structures	20
5.1	More on Lists	20
5.1.1	Using Lists as Stacks	21
5.1.2	Using Lists as Queues	21
5.1.3	Functional Programming Tools	22

5.1.4	List Comprehensions	23
5.2	The <code>del</code> Statement	25
5.3	Tuples and Sequences	25
5.4	Sets	26
5.5	Dictionaries	27
5.6	Looping Techniques	28
5.7	More on Conditions	29
5.8	Comparing Sequences and Other Types	29
6	Modules	30
6.1	More on Modules	31
6.1.1	Executing modules as scripts	31
6.1.2	The Module Search Path	32
6.1.3	“Compiled” Python files	32
6.2	Standard Modules	33
6.3	The <code>dir()</code> Function	33
6.4	Packages	34
6.4.1	Importing <code>*</code> From a Package	35
6.4.2	Intra-package References	36
6.4.3	Packages in Multiple Directories	36
7	Input and Output	36
7.1	Fancier Output Formatting	36
7.1.1	Old string formatting	39
7.2	Reading and Writing Files	40
7.2.1	Methods of File Objects	40
7.2.2	Saving structured data with <code>json</code>	41
8	Errors and Exceptions	42
8.1	Syntax Errors	42
8.2	Exceptions	43
8.3	Handling Exceptions	43
8.4	Raising Exceptions	45
8.5	User-defined Exceptions	45
8.6	Defining Clean-up Actions	47
8.7	Predefined Clean-up Actions	47
9	Classes	48
9.1	A Word About Names and Objects	48
9.2	Python Scopes and Namespaces	48
9.3	A First Look at Classes	49
9.3.1	Class Definition Syntax	50
9.3.2	Class Objects	50
9.3.3	Instance Objects	51
9.3.4	Method Objects	52
9.3.5	Class and Instance Variables	52
9.4	Random Remarks	53
9.5	Inheritance	54
9.5.1	Multiple Inheritance	55
9.6	Private Variables and Class-local References	55
9.7	Odds and Ends	56
9.8	Exceptions Are Classes Too	56
9.9	Iterators	57
9.10	Generators	58
9.11	Generator Expressions	59

10 Brief Tour of the Standard Library	59
10.1 Operating System Interface	59
10.2 File Wildcards	60
10.3 Command Line Arguments	60
10.4 Error Output Redirection and Program Termination	60
10.5 String Pattern Matching	60
10.6 Mathematics	60
10.7 Internet Access	61
10.8 Dates and Times	61
10.9 Data Compression	62
10.10 Performance Measurement	62
10.11 Quality Control	62
10.12 Batteries Included	63
11 Brief Tour of the Standard Library — Part II	64
11.1 Output Formatting	64
11.2 Templating	65
11.3 Working with Binary Data Record Layouts	66
11.4 Multi-threading	66
11.5 Logging	67
11.6 Weak References	67
11.7 Tools for Working with Lists	68
11.8 Decimal Floating Point Arithmetic	69
12 What Now?	70
13 Interactive Input Editing and History Substitution	70
13.1 Line Editing	70
13.2 History Substitution	71
13.3 Key Bindings	71
13.4 Alternatives to the Interactive Interpreter	72
14 Floating Point Arithmetic: Issues and Limitations	72
14.1 Representation Error	74

1 Whetting Your Appetite

1.1 Some applications of Python

- Automate works on computer
- Help software engineers develop and test more quickly
- ...

1.2 The features of Python

- High-level language
High-level more general data types
- Allow to split program into modules
- Interpreted language
 - No necessary compilation and linking
 - The interpreter can be used interactively
- Shorter programs than equivalent C, C++ or Java programs
 - high-level data types
 - statement grouping is done by indentation instead of brackets
 - no variable or argument declarations are necessary
- Extensible

1.3 The origin of the name "Python"

"Monty Python's Flying Circus", the skits on BBC.

2 Using the Python Interpreter

2.1 Invoking the Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python`, and usually the `/usr/local/bin` has been put in the Unix shell's search path. Type command `python` to the shell to invoke the Python interpreter.

Type an end-of-file character (`Ctrl-D` on Unix, `Ctrl-Z` on Windows) to exit the interpreter. If that doesn't work, type command `quit()` to the shell.

A second way of starting the interpreter is `python -c 'command [arg] ...'`, which executes the statement(s) in command.

Some Python modules are also useful as scripts. These can be invoked using `python -m module [arg] ...`, which executes the source file for modules as if you had spelled out its full name on the command line.

2.1.1 Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the `argv` variable in the `sys` module. You can access this list by executing `import sys`. The length of the list is at least one; when no script and no arguments are given, `sys.argv[0]` is an empty string. When the script name is given as `'-'` (meaning standard input), `sys.argv[0]` is set to `'-'`. When `-c` command is used, `sys.argv[0]` is set to `'-c'`. When `-m` module is used, `sys.argv[0]` is set to the full name of the located module. Options found after `-c` command or `-m` module are not consumed by the Python interpreter's option processing but left in `sys.argv` for the command or module to handle.

2.1.2 Interactive Mode

- *primary prompt*: `>>>`

```
Python 2.7.10 (default, Feb  7 2017, 00:08:15)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- *secondary prompt*: `...`

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

2.2 The Interpreter and Its Environment

2.2.1 Source Code Encoding

By default, Python source files are treated as encoded in UTF-8.

To declare an encoding other than the default one, a special comment line should be added as the first line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

where *encoding* is one of the valid codes supported by Python.

3 An Informal Introduction to Python

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`).

Comments in Python start with the hash character `#`, and extend to the end of the physical line. A hash character in a string literal is just a hash character.

```
# this is the first comment
spam = 1    # and this the second comment
            # and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Using Python as a Calculator

3.1.1 Numbers

```
>>> 2 + 2
4
>>> 50 - 5 * 6
20
>>> (50 - 5.0 * 6) / 4
5.0
>>> 8 / 5.0
1.6
```

The integer numbers (e.g. 2, 4, 20) have type `int`, the ones with a fractional part (e.g. 5.0, 1.6) have type `float`.

The return type of a division (`/`) operation depends on its operands. If both operands are of type `int`, floor division is performed and an `int` is returned. If either operand is a float, classic division is performed and a float is returned. The `//` operator is also provided for doing floor division no matter what the operands are. The remainder can be calculated with the `%` operator:

```
>>> 17 / 3      # int / int -> int
5
>>> 17 / 3.0    # int / float -> float
5.666666666666667
>>> 17 // 3.0   # explicit floor division discards the fractional part
5.0
>>> 5 * 3 + 2   # result * divisor + remainder
17
```

With Python, it is possible to use ****** operator to calculate powers:

```
>>> 5 ** 2      # 5 squared
25
>>> 2 ** 7      # 2 to the power of 7
128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no results is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not "defined" (assigned a value), trying to use it will give you an error:

```
>>> n          # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

In interactive mode, the last printed expression is assigned to the variable `_`.

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

The variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to `int` and `float`, Python supports other types of numbers, such as `Decimal` and `Fraction`. Python also has built-in support for complex numbers, and uses the `j` or `J` suffix to indicate the imaginary part (e.g. `3 + 5j`).

3.1.2 Strings

Python can also manipulate strings, which should be enclosed in single quotes ('...') or double quotes ("...") with the same result. `\` can be used to escape quotes.

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> 'doesn\'t' # use \' to escape the single quote...
'doesn't'
>>> "doesn't" # ... or use double quotes instead
'doesn't'
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

The `print` statement produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print '"Isn\'t," she said.'
"Isn't," she said.
>>> s = 'First line. \nSecond line.' # \n means newline
>>> s # without print, \n is included in the output
'First line. \nSecond line.'
>>> print s # with print, \n produces a new line
First line.
Second line.
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>> print 'C:\some\name' # here \n means newline!
C:\some
ame
>>> print r'C:\some\name' # note the r before the quote
C:\some\name
```

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End of lines are automatically included in the string, but it's possible to prevent this by adding a `\` at the end of the line:

```
>>> print """
... Usage: thingy [OPTIONS]
...     -h             Display this usage message
...     -H hostname    Hostname to connect to
... """

Usage: thingy [OPTIONS]
    -h             Display this usage message
    -H hostname    Hostname to connect to

>>> print """\ # prevent the initial newline
... Usage: thingy [OPTIONS]
...     -h             Display this usage message
...     -H hostname    Hostname to connect to
... """

Usage: thingy [OPTIONS]
    -h             Display this usage message
    -H hostname    Hostname to connect to
```

Strings can be concatenated (glued together) with + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
... 3 * 'un' + 'ium'
'unununium'
```

Two or more string literals next to each other are automatically concatenated:

```
>>> 'Py' 'thon'
'Python'
```

This only works with two literals though, not with variables or expressions. If you want to concatenate variables or a variable and a literal, use +.

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and string literal
File "<stdin>", line 1
    prefix 'thon' # can't concatenate a variable and string literal
    ^
SyntaxError: invalid syntax
```

```
>>> prefix + 'thon'
'Python'
```

This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several string within parentheses '
...         'to have them joined together.')
>>> text
'Put several string within parentheses to have them joined together.'
```

Strings can be indexed(subscripted), with the first character having index 0. There is no separate character type; a character is simple a string a size one:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indices may also be negative nubmers, to start counting from the right:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Since indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.


```
>>> word[:2]      # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]      # character from the position 4 (included) to the end
'on'
>>> word[-2:]     # character from the second-last (included) to the end
'on'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of `n` characters has index `n`, for example:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Attempting to use an index that is too large will result in an error:

```
>>> word[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indices are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to a indexed position in the string results in an error:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

```
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
```

34

3.1.3 Unicode Strings

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts.

Creating Unicode strings in Python is just as simple as creating normal strings:

```
>>> u'Hello World!'
u'Hello World!'
```

The small 'u' in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python *Unicode-Escape* encoding. The following example shows how:

```
>>> u'Hello\u0020World!'
u'Hello World!'
```

The escape sequence `\u0020` indicates to insert the Unicode character with the ordinal value 0x0020 (the space character) at the given position.

3.1.4 Lists

Python knows a number of compound data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in sequence type), lists can be indexed and sliced:

```
>>> squares[0]      # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]    # slicing returns a new list
[9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also supports operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the `append()` method:

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # add the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
... letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
... letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
... letters[:] = []
>>> letters
[]
```

The built-in function `len()` also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
>>> x[1]
[1, 2, 3]
```

3.2 First Steps Towards Programming

We can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1      # multiple assignment
>>> while b < 10:    # loop as long as the condition remains true
...     print b
...     a, b = b, a + b
...
1
1
2
3
5
8
```

In Python, like in C, any non-zero integer value is true; zero is false. The condition of the `while` loop may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences

are false. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).

The *body* of loop is *indented*: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line.

The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write in the way it handles multiple expressions and strings. Strings are pointer without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256 * 256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a + b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

4 More Control Flow Tools

4.1 if Statements

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

4.2 for Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print w, len(w)
...
cat 3
window 6
defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop, it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
>>> for w in words[:]:
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 The range() function

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)    # the given end point is never part of the generated list
[5, 6, 7, 8, 9]
>>> range(0, 10, 3) # '3' here is the increment('step'), it can be negative
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

4.4 break and continue Statements, and else Clauses on Loops

The `break()` statement, like in C, breaks out of the innermost enclosing `for` or `while` loop.

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n / x
...             break
...         else:
...             # loop fell through without finding a factor
...             print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs.

The `continue` statement, also borrowed from C, continues with the next iteration of the loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
...
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5 `pass` Statements

The `pass` statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass    # Busy-wait for keyboard interrupt (Ctrl + C)
...
^CTraceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyboardInterrupt
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptClass:
...     pass
...
```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass    # Remember to implement this!
...
```

4.6 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a + b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's a good practice to include docstrings in code that you write, so make a habit of it.

The **execution** of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a userdefined function. This value can be assigned to another name which can be used as a function. This serves as a general renaming mechanism:

```
>>> fib
<function fib at 0x106cd8140>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (It's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see if you really want to using `print`:

```
>>> fib(0)
>>> print fib(0)
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100)
>>> f100
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- The statement `result.append(a)` calls a method of the list object `result`. A method is a function that 'belongs' to an object and is named `obj.methodname`, where `obj` is some object (this may be an expression), and `methodname` is the name of a method that is defined by the object's type. The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.7 More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.7.1 Default Arguments Values

The most useful form is to specify a default value for one or more arguments. The default arguments can't be followed by non-default arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
>>> def ask_ok(prompt, retries = 4, complaint = "Yes or no, please!"):
...     while True:
...         ok = raw_input(prompt)
...         if ok in ('y', 'ye', 'yes'):
...             return True
...         if ok in ('n', 'no', 'nop', 'nope'):
...             return False
...         retries = retries - 1
...         if retries < 0:
...             raise IOError('refusenik user')
...         print complaint
... 
```

This function can be called in several ways:

- giving only the mandatory argument:
`ask_ok('Do you really want to quit?')`
- giving one of the optional arguments:
`ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments:
`ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

The keyword `in` can test whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope. In other words, default parameter values are always evaluated when, and only when, the `def` statement they belong to is executed, so that

```
>>> i = 5
>>> def f(arg = i):
...     print arg
...
>>> f()
5
>>> i = 6
>>> f()
5
```

Important warnings: The default value is evaluated only once. ¹ This makes a difference when the default is a mutable object ² such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
>>> def f(a, L = []):
...     L.append(a)
...     return L
... 
```

¹<http://effbot.org/zone/default-values.htm>

²<https://codehabitude.com/2013/12/24/python-objects-mutable-vs-immutable/>


```
>>> print f(1)
[1]
>>> print f(2)
[1, 2]
>>> print f(3)
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
>>> def f(a, L = None):
...     if L is None:
...         L = []
...     L.append(a)
...     return L
...
>>> print f(1)
[1]
>>> print f(2)
[2]
>>> print f(3)
[3]
```

4.7.2 Keyword Arguments

Functions can also be called using keyword arguments of the form `kwarg = value`.

```
>>> def parrot(voltage, state = 'a stiff', action = 'vroom', type = 'Norwegian Blue'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it."
...     print "-- Lovely plumage, the", type
...     print "-- It's", state, "!"
...
```

This function can be called in any of the following ways:

```
parrot(1000)                                # 1 positional argument
parrot(voltage=1000)                        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')   # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)    # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()                                # required argument missing
parrot(voltage=5.0, 'dead')             # non-keyword argument after a keyword argument
parrot(110, voltage=220)                 # duplicate value for the same argument
parrot(actor='John Cleese')             # unknown keyword argument
```

When a final formal parameter of the form `**name` is present, it receives a dictionary containing all keyword arguments except for those corresponding to a formal parameter. This may be contained with a formal parameter of the form `*name` which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
>>> def cheeseshop(kind, *arguments, **keywords):
...     print "-- Do you have any", kind + "?"
...     print "-- I'm sorry, we're all out of", kind
...     for arg in arguments:
...         print arg
```

```

...     print "-" * 40
...     keys = sorted(keywords.keys())
...     for kw in keys:
...         print kw, ":", keywords[kw]
...
>>> cheeseshop("Limburger", "It's very runny, sir.",
...             "It's really very, VERY runny, sir.",
...             shopkeeper = 'Michael Palin',
...             client = "John Cleese",
...             sketch = "Cheese Shop Sketch")
-- Do you have any Limburger?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch

```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

4.7.3 Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instances, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```

>>> range(3, 6)
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)
[3, 4, 5]

```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```

>>> def parrot(voltage, state = 'a stiff', action = 'vroom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state + "!"
...
>>> d = {"voltage" : "four million", "state" : "blledin' demised", "action" : "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's blledin' demised!

```

4.7.4 Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b : a + b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```

>>> def make_incrementor(n):
...     return lambda x : x + n      # return a function
...
>>> f = make_incrementor(42)

```

```
>>> f(0)
42
>>> f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key = lambda pair : pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.5 Documentation Strings

There are emerging conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object’s purpose. For brevity, it should not explicitly state the object’s name or type, since these are available by other means (except if the name happens to be a verb describing a function’s operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object’s calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. Whitespace “equivalent” to this indentation is then stripped from the start of all lines of the string.

Here is an example of a multi-line docstring:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8 Intermezzo: Coding Style

Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, PEP 8³ has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted:

- Use 4-space indentation, and no tabs.
- Wrap lines so that they don’t exceed 79 characters.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.

³<https://www.python.org/dev/peps/pep-0008>

- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores` for functions and methods. Always use `self` as the name for the first method argument.
- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

5 Data Structures

5.1 More on Lists

All of the methods of list objects:

- `list.append(x)`
Add an item to the end of the list; equivalent to `a[len(a):] = x`.
- `list.extend(L)`
Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`
- `list.insert(i, x)`
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
- `list.remove(x)`
Remove the first item from the list whose value is `x`.
- `list.pop([i])`
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)
- `list.index(x)`
Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.
- `list.count(x)`
Return the number of times `x` appears in the list.
- `list.sort(cmp = None, key = None, reverse = False)`
Sort the items of the list in place (the arguments can be used for sort customization, see `sorted`⁴ for their explanation).
- `list.reverse()`
Reverse the elements of the list, in place.

An example that uses most of the list methods:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
```

⁴<https://docs.python.org/2.7/library/functions.html#sorted>

```

1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]

```

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed — they return the default `None`. This is a design principle for all mutable data structures in Python.

5.1.1 Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```

>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]

```

5.1.2 Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all the other elements have to be shifted by one).

To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```

>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'

```

```
>>> queue                                     # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Functional Programming Tools

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

`filter(function, sequence)` returns a sequence consisting of those item from the sequence for which `function(item)` is true. If `sequence` is a `str`, `unicode` or `tuple`, the result will be of the same type; otherwise, it is always a list. For example, to compute a sequence of numbers divisible by 3 or 5:

```
>>> def f(x) : return x % 3 == 0 or x % 5 == 0
...
>>> filter(f, range(2, 25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```
>>> def cube(x) : return x * x * x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or `None` if some sequence is shorter than another). For example:

```
>>> seq = range(8)
>>> def add(x, y) : return x + y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` returns a single value constructed by calling the binary function `function` on the first two items of the sequence, then on the result and the next items, and so on. For examples, to compute the sum of the numbers 1 through 10:

```
>>> def add(x, y) : return x + y
...
>>> reduce(add, range(1, 11))
55
>>> reduce(add, range(1,2))
1
>>> reduce(add, range(1,1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
```

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```
>>> def sum(seq):
...     def add(x, y) : return x + y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Don't use this example's definition of `sum()`: since summing nubmers is such a common need, a built-in function `sum(sequence)` is already provided, and works exactly like this.

5.1.4 List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is teh result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can obtain the same result with:

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This is also equivalent to `squares = map(lambda x : x ** 2, range(10))`, but it's more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
>>> combs = []
>>> for x in [1, 2, 3]:
...     for y in [3, 1, 4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

If the expression is a tuple (e.g. the `(x, y)` in the previous example), it must be parenthesized.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
... [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
... [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
... [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
... freshfruit = [' banana', ' loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

```

>>> # create a list of 2-tuples like (number, square)
... [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
... [x, x**2 for x in range(6)]
      File "<stdin>", line 2
        [x, x**2 for x in range(6)]
            ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
... vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

```

List comprehension can contain complex expressions and nested functions:

```

>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']

```

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3×4 matrix implemented as a list of 3 lists of length 4:

```

>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12]
... ]

```

The following list comprehension will transpose rows and columns:

```

>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

```

As we saw in the previous section, the nested listcomp is evaluated in the context of the `for` that follows it, so this example is equivalent to:

```

>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

```

which, in turn, is the same as:

```

>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

```

In the real world, you should prefer built-in functions to complex flow statements. The `zip()` function would do a great job for this use case:

```

>>> zip(*matrix) # use * to unpack the arguments from matrix
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

```


5.2 The del Statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` can also be used to delete entire variables. Referencing the name `a` hereafter is an error (at least until another value is assigned to it):

```
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

5.3 Tuples and Sequences

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing (or even by attribute in the case of `namedtuples`). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item

is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
>>> empty = ()
>>> singleton = 'hello',      # <-- note trailing comma
>>> len(singleton)
1
>>> singleton
('hello',)
>>> singleton = 'hello'      # the type of singleton here is 'str'
>>> len(singleton)
5
>>> singleton
'hello'
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

```
>>> x, y, z = t
>>> x
12345
>>> y
54321
>>> z
'hello!'
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4 Sets

Python also includes a data type for *sets*. A set is an unsorted collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{}`; the latter creates an empty dictionary.

Here is a brief demonstration:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)      # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit        # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> b
set(['a', 'c', 'z', 'm', 'l'])
>>> a - b
```

```

set(['r', 'b', 'd'])
>>> a | b
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
>>> a & b
set(['a', 'c'])
>>> a ^ b
set(['b', 'd', 'm', 'l', 'r', 'z'])

```

Similarly to list comprehensions, set comprehensions are also supported:

```

>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])

```

5.5 Dictionaries

Another useful data type built into Python is the *dictionary*. Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable types; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contain any mutable object either directly or indirectly, it cannot be used as a key. You can't use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as an unsorted set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sorted()` function to it). To check whether a single key is in the dictionary, use `in` keyword.

Here is a small example using a dictionary:

```

>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'irv': 4127, 'guido': 4127}
>>> tel.keys()
['jack', 'irv', 'guido']
>>> 'guido' in tel
True
>>> 'tom' in tel
False

```

The `dict()` constructor builds dictionaries from sequences of key-value pairs:

```

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}

```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```

>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}

```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6 Looping Techniques

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the `zip()` function.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}? It is {1}'.format(q, a)
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the `reversed()` function.

```
>>> for i in reversed(xrange(1, 10, 2)):
...     print i
...
9
7
5
3
1
```

To loop over a sequence in sorted order, use the `sorted()` function which returns a new sorted list while leaving the source unaltered.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
>>> basket
['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
```

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
```

```
...
gallahad the pure
robin the brave
```

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51, 7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51, 7, 55.3, 52.5, 47.8]
```

5.7 More on Conditions

The conditions used in `while` and `if` statements can contain any operators, not just comparisons.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, `a < b == c` tests whether `a` is less than `b` and moreover `b` equals `c`.

Comparisons may be combined using the Boolean operators `and` and `or`, and the outcome of a comparison (or of any other Boolean expression) may be negated with `not`. These have lower priorities than comparison operators; between them, `not` has the highest priority and `or` the lowest, so that `A and B not B or C` is equivalent to `(A and (not B)) or C`. As always, parentheses can be used to express the desired composition.

The Boolean operators `and` and `or` are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if `A` and `C` are true but `B` is false, `A and b and C` does not evaluate the expression `C`. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

```
>>> 3 and 5 and 6 and 7
7
>>> 3 or 4 or 5
3
>>> 0 or 6 or 7
6
```

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing `=` in an expression when `==` was intended.⁵

5.8 Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome

⁵<https://stackoverflow.com/questions/2603956/can-we-have-assignment-in-a-condition>

of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequence compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

```
>>> (1, 2, 3) < (1, 2, 4)
>>> [1, 2, 3] < [1, 2, 4]
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
>>> (1, 2, 3, 4) < (1, 2, 4)
>>> (1, 2) < (1, 2, -1)
>>> (1, 2, 3) == (1.0, 2.0, 3.0)
>>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc. Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

6 Modules

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definitions into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
fibo
```

If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement. (They are also run if the file is executed as a script.)

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can touch a module's global variables with the name notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all `import` statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the `import` statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore(_).

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Note: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter — or, if it's just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

6.1.1 Executing modules as scripts

When you run a Python module with `python fibo.py <arguments>` the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

```
duangdeiMac:6 duang$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
>>> import fibo
>>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

6.1.2 The Module Search Path

When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories given by the variable `sys.path`. `sys.path` is initialized from these locations:

- the directory containing the input script (or the current directory)
- `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable `PATH`)
- the installation-dependent default

After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended.

6.1.3 “Compiled” Python files

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called `spam.pyc` exists in the directory where `spam.py` is found, this is assumed to contain an already-“byte-compiled” version of the module `spam`. The modification time of the version of `spam.py` used to create `spam.pyc` is recorded in `spam.pyc`, and the `.pyc` file is ignored if these don’t match.

Normally, you don’t need to do anything to create the `spam.pyc` file. Whenever `spam.py` is successfully compiled, an attempt is made to write the compiled version to `spam.pyc`. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting `spam.pyc` file will be recognized as invalid and thus ignored later. The contents of the `spam.pyc` file are platform independent, so a Python module directory can be shared by machines of different architectures.

Some tips for experts:

- When the Python interpreter is invoked with the `-o` flag, optimized code is generated and stored in `.pyo` files. The optimizer currently doesn’t help much; it only removes `assert` statements. When `-o` is used, all bytecode is optimized; `.pyc` files are ignored and `.py` files are compiled to optimized bytecode.
- Passing two `-o` flags to the Python interpreter (`-oo`) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only `__doc__` strings are removed from the bytecode, resulting in more compact `.pyo` files. Since some programs may rely on having these available, you should only use this option if you know what you’re doing.
- A program doesn’t run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from `.py` file; the only thing that’s faster about `.pyc` or `.pyo` files is the speed which they are loaded.
- When a script is run by giving its name of the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a `.pyc` or `.pyo` file directly on the command line.
- It is possible to have a file called `spam.pyc` (or `spam.pyo` when `-o` is used) without a file `spam.py` for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
- The module `compileall` can create `.pyc` files (or `.pyo` files when `-o` is used) for all modules in a directory.

6.2 Standard Modules

Python comes with a library of standard modules, described in a separated document, the Python Library Reference. Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'...'
>>> sys.ps3
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
>>> import sys
>>> sys.path.append('/ufs/gudio/lib/python')
```

6.3 The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__egginsert', '__excepthook__', '__name__', '__package__',
'__plen', '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_getframe', '_mercurial', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode',
'exc_clear', 'exc_info', 'exc_type', 'excepthook', 'exec_prefix', 'executable', 'exit',
'flags', 'float_info', 'float_repr_style', 'getcheckinterval', 'getdefaultencoding',
'getdlopenflags', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount',
'getsizeof', 'gettrace', 'hexversion', 'last_traceback', 'last_type', 'last_value',
'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'py3kwarning', 'setcheckinterval',
'setdlopenflags', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin',
'stdout', 'subversion', 'version', 'version_info', 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'fib', 'fibo']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError',
'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception',
'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented', 'NotImplementedError',
'OSError', 'OverflowError', 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any', 'apply',
'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod',
'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long',
'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'quit', 'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 Packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other's module names.

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you'll be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

<code>sound/</code>	Top-level package
<code>__init__.py</code>	Initialize the sound package
<code>formats/</code>	Subpackage for file format conversions
<code>__init__.py</code>	
<code>wavread.py</code>	
<code>wavwrite.py</code>	
<code>aiffread.py</code>	
<code>aiffwrite.py</code>	
<code>auread.py</code>	
<code>auwrite.py</code>	
<code>...</code>	
<code>effects/</code>	Subpackage for sound effects
<code>__init__.py</code>	
<code>echo.py</code>	
<code>surround.py</code>	
<code>reverse.py</code>	
<code>...</code>	
<code>filters/</code>	Subpackage for filters
<code>__init__.py</code>	

```

equalizer.py
vocoder.py
karaoke.py
...

```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay = 0.7, atten = 4)
```

An alternative way of importing the submodule is:

```
from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
echo.echofilter(input, output, delay = 0.7, atten = 4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
echofilter(input, output, delay = 0.7, atten = 4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or a function or variable defined in the previous item.

6.4.1 Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing `*` from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
__all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the `sound` package.

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined by in the `sound.effects` package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

6.4.2 Intra-package References

The submodules often needs to refer to each other. For example, the `surround` module might use the `echo` module. In fact, such references are so common that the `import` statement first looks in the containing package before looking in the standard module search path. Thus, the `surround` module can simply use `import echo` or `from echo import echofilter`. If the imported module is not found in the current package (the package of which the current module is a submodule), the `import` statement looks for top-level module with the given name.

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the `from module import name` form of import statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always “`__main__`”, modules intended for use as the main module of a Python application should always use absolute imports.

6.4.3 Packages in Multiple Directories

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package’s `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

7 Input and Output

7.1 Fancier Output Formatting

So far we’ve encountered two ways of writing values: *expression statements* and the `print` statement. (A third way is using the `write()` method of file objects; the standard output file can be referenced as

`sys.stdout.)`

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string types have some methods that perform useful operations for padding strings to a given columns width; these will be discussed shortly. The second way is to use the `str.format()` method.

The `string` module contains a `Template` class which offers yet another way to substitute values into strings.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to string: pass it to `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, which `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs'))))
"(32.5, 40000, ('spam', 'eggs'))"
```

Here are two ways to write a table of squares and cubes:

```
>>> for x in range(1, 11):
...     # S.rjust(width) returns S right-justified in a string of length width
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
1    1    1
2    4    8
3    9   27
4   16   64
5   25  125
6   36  216
7   49  343
8   64  512
9   81  729
```

```
10 100 1000
```

```
>>> for x in range(1, 11):
...     print '{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x)
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```

(Note that in the first example, one space between each column was added by the way `print` works: it always adds spaces between its arguments.)

This example demonstrates the `str.rjust()` method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column layout but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill(width)`, which pads a numeric string on the left with zeros. It understands about plus and minus signs. The original string is returned if width is less than or equal to `len(str)`.

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Basic usage of `str.format()` method looks like this:

```
>>> print 'We are the {} whos say "{}!"'.format('knights', 'Ni')
We are the knights whos say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets refers to the position of the object passed into the `str.format()` method.

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the arguments.

```
>>> print 'This {food} is {adjective}'.format(
...     food = 'spam', adjective = 'absolutely horrible')
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrary combined:

```
>>> print 'The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other = 'Georg')
The story of Bill, Manfred, and Georg.
```

```
>>> print 'The story of {0}, {1}, and {other}.'.format('Bill', other = 'Georg', 'Manfred')
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

'!s' (apply `str()`) and '!r' (apply `repr()`) can be used to convert the value before it is formatted.

```
>>> import math
>>> print 'The value of PI is approximately {}'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!s}'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

An optional ':' and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

```
>>> print 'The value of PI is approximately {0:.3f}'.format(math.pi)
The value of PI is approximately 3.142.
```

Passing an integer after the ':' will cause that field to be a minimum number of characters wide. This is useful for making tables pretty.

```
>>> for name, phone in table.items():
...     print '{0:10} ==> {1:10d}'.format(name, phone)
...
Dcab      ==>      7678
Sjoerd    ==>      4127
Jack      ==>      4098
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets '[]' to access the keys.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the `**` notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

7.1.1 Old string formatting

The `%` operator can also be used for string formatting. It interprets the left argument much like a `sprintf()`-style format string to be applied to the right argument, and returns the string resulting from this formatting operation. For example:

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

7.2 Reading and Writing Files

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 0x107dfc5d0>
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

On Windows, `'b'` appended to the mode opens the file in binary mode, so there are also modes like `'rb'`, `'wb'`, and `'r+b'`. Python on Windows makes a distinction between text and binary files; the end-of-line character in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn't hurt to append a `'b'` to the mode, so you can use it platform-independently for all binary files.

7.2.1 Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character `'\n'` is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> # the file doesn't end in a newline
... f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file.\n'
>>> f.readline()
''
```

```
>>> # the file ends in a newline
... f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file.\n'
>>> f.readline()
'\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:


```
>>> for line in f:
...     print line,
...
This is the first line of the file.
Second line of the file.
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`. `f.write(string)` writes the contents of *string* to the file, returning `None`.

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)
>>> f.read(1)
'5'
>>> f.seek(-3, 2)
>>> f.read(1)
'd'
```

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

It is a good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile', 'r') as f:
...     read_data = f.read()
...
>>> f.closed
True
>>> read_data
'0123456789abcdef'
```

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

7.2.2 Saving structured data with json

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like

'123' and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

Note: The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a file. So if `f` is a file object opened for writing, we can do this:

```
>>> x = json.dumps([1, 'simple', 'list'])
>>> f = open('workfile', 'r+')
>>> json.dump(x, f)
>>> f.seek(0)
>>> f.readlines()
['[1, \\\"simple\\\"\", \\\"list\\\""]']
>>> f.close()
```

To decode the object again, if `f` is a file object which has been opened for reading:

```
>>> with open('workfile', 'r') as f:
...     x = json.load(f)
...
>>> x
u'[1, "simple", "list"]'
```

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

See also: `pickle` - the pickle module

Contrary to JSON, *pickle* is a protocol which allows the serialization of arbitrary complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.

8 Errors and Exceptions

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

8.1 Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print 'Hello world'
File "<stdin>", line 1
    while True print 'Hello world'
                ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the keyword `print`, since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 Exceptions

Even if statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam * 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for userdefined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Built-in Exceptions lists the built-in exceptions and their meanings.

8.3 Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using **Control-C** or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
...
Please enter a number: j
Oops! That was no valid number. Try again...
```

The `try` statement works as follows:

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the except clause is executed, and then execution continues after the `try` statement.

- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statement; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A try statement may have more than one except clause, to specify handles for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Note that the parentheses around this tuple are required, because `except ValueError, e:` was the syntax used for what is normally written as `except ValueError as e:` in modern Python (described below). The old syntax is still supported for backwards compatibility. This means `except RuntimeError, TypeError` is not equivalent to `except (RuntimeError, TypeError):` but to `except RuntimeError as TypeError:` which is not what you want.

The last except clause may omit exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise exception (allowing a caller to handle the exception as well):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

The try ... except statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

The use of the else clause is better than adding additional code to the try clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the try ... except statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`.

One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> try:
...     raise Exception('spam', 'eggs')
```

```

... except Exception as inst:
...     print type(inst)      # the exception instance
...     print inst.args      # arguments stored in .args
...     print inst           # __str__ allows args to be printed directly
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

If an exception has an argument, it is printed as the last part (‘detail’) of the message for unhandled exceptions.

Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

```

>>> def this_fails():
...     x = 1 / 0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero

```

8.4 Raising Exceptions

The **raise** statement allows the programmers to force a specified exception to occur. For example:

```

>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere

```

The sole argument to **raise** indicates the exception to be raised. This must be either an exception instance or an exception (a class that derives from **Exception**):

```

>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise      # re-raise the exception to see if there is a exception
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere

```

8.5 User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see **Classes** for more about Python classes). Exceptions should typically be derived from the **Exceptions** class, either directly or indirectly. For example:

```

>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: 'oops!'

```

In this example, the default `__init()` of `Exception` has been overridden. The new behavior simply creates the `value` attribute. This replaces the default behavior of creating the `args` attribute.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```

class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exceptions raised for errors in the input.

    Attributes:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not allowed.

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """

    def __init__(self, prev, next, msg):
        self.prev = prev
        self.next = next
        self.msg = msg

```

Most exceptions are defined with names that end in “Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define.

8.6 Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

A *finally clause* is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed “on the way out” when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example (having `except` and `finally` clauses in the same `try` statement works as of Python 2.5):

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clauses"
...
>>> divide(2, 1)
result is 2
executing finally clauses
>>> divide(2, 0)
division by zero!
executing finally clauses
>>> divide("2", "1")
executing finally clauses
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

8.7 Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print line,
```

The problem with this code is that it leaves the file open for an interminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
with open("myfile.txt") as f:
    for line in f:
        print line,
```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.

9 Classes

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are *public* (except see below Private Variables and Class-local References), and all member functions are *virtual*. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

9.1 A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

9.2 Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespace is that there is absolutely no relation between names in different namespaces; for instance, two different modules

may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module’s attribute and the global names defined in the module: they share the same namespace!

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `__builtin__`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespace are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module’s global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module’s global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module’s namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module’s namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definitions is evolving towards static name resolution, at “compile” time, so don’t rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that — if no `global` statement is in effect — assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)

9.3 A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

9.3.1 Class Definition Syntax

The simplest form of class definition looks like this:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definitions in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the `end`), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definitions; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definitions was entered) is reinstalled, and the class object is bound here to the class name given in the class definitions header (`ClassName` in the example).

9.3.2 Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

If you write the definition of the class `MyClass` in script `MyClass.py`, then you can use it by command `import MyClass`. However, when I tried to use it in the interpreter with `x = MyClass()`, I got the following error message:

```
>>> x = MyClass()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'module' object is not callable
```

That's because the clause `import MyClass` is importing a *module* named `MyClass` instead of the class `MyClass`.⁶ So we can call the class `MyClass` by the following way:

```
>>> import MyClass
>>> x = MyClass.MyClass()
```

Alternatively, we can use `from MyClass import MyClass`, like the following code:

⁶<https://stackoverflow.com/questions/16780510/>

```

>>> from MyClass import MyClass
>>> x = MyClass()
>>> x.i
12345
>>> x.f()
'hello world'

```

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class `x = MyClass()` creates a new *instance* of the class and assigns this object to the local variable `x`).

This instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```

def __init__(self):
    self.data = []

```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by: `x = MyClass()`

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```

>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)

```

9.3.3 Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```

>>> x.counter = 1
>>> while x.counter < 10:
...     x.counter = x.counter * 2
...
>>> print x.counter
16
>>> del x.counter

```

The other kind of instance attribute reference is a *method*. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called `append`, `insert`, `remove`, `sort` and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

9.3.4 Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print xf()
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. **In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.**

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When an instance attribute is referenced that isn't a data attribute, its class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

9.3.5 Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:
    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind
'canine'
>>> e.kind
'canine'
>>> d.name
'Fido'
>>> e.name
'Buddy'
```

As discussed in *A Word About Names and Objects*, shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
class Dog:

    tricks = []             # mistaken use of a class variable
```

```

def __init__(self, name):
    self.name = name

def add_trick(self, trick):
    self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks          # unexpectedly shared by all dogs
['roll over', 'play dead']

```

Correct design of the class should use an instance variable instead:

```

class Dog:
    def __init__(self, name):
        self.name = name
        self.tricks = []    # create a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']

```

9.4 Random Remarks

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Data attributes may be referenced by methods as well as by ordinary users (“clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browse* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```

# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g

```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attribute of the `self` argument:

```

class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def add_twice(self, x):
        self.add(x)
        self.add(x)

```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

9.5 Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

```

class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>

```

The name `BaseClassName` must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module: `class DerivedClassName(modname.BaseClassName):`

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively *virtual*.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is `False` since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).

9.5.1 Multiple Inheritance

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first — searching `Base2` and `Base3` before the base classes of `Base1` — looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no difference between direct and inherited attributes of `Base1`.)

For new-style classes, the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the `super` call found in single-inheritance languages.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance.

9.6 Private Variables and Class-local References

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped.

This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update          # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec`, `eval()` or `execfile()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()`, and `delattr()`, as well as when referencing `__dict__` directly.

9.7 Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal “record” or C “struct”, building together a few named data items. An empty class definition will do nicely:

```
class Employee:
    pass

john = Employee()    # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.

9.8 Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:


```
raise Class, instance
raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

```
raise instance.__class__, instance
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
class B:
    pass

class C(B):
    pass

class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Note that if the except clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching except clause is triggered.

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.

9.9 Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one': 1, 'two': 2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
```

```

>>> it
<iterator object at 0x109ceb950>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then `__iter__()` can just return `self`:

```

>>> from reverse import Reverse
>>> rev = Reverse('spam')
>>> iter(rev)
<reverse.Reverse instance at 0x10dcd1488>
>>> for char in rev:
...     print char
...
m
a
p
s

```

9.10 Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```

def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g

```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

Another key feature is that the local variable and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

9.11 Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of brackets. These expressions are designed for situations where the generators is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```
>>> sum(i*i for i in range(10))           # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x, y in zip(xvec, yvec))   # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word for line in page for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

10 Brief Tour of the Standard Library

10.1 Operating System Interface

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()           # Return the current working directory
'/Users/duang/Documents/learning-notes/Programming/Python/the-python-tutorial/9'
>>> os.chdir('.')        # Change the current working directory
>>> os.getcwd()
'/Users/duang/Documents/learning-notes/Programming/Python/the-python-tutorial'
>>> os.system('mkdir today') # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('bag.py', 'bag_copy.py')
>>> shutil.move('bag_copy.py', '../bag_copy.py')
```

10.2 File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['bag.py', 'dog.py', 'employee.py', 'exception_classes.py', 'mapping.py', 'my_class.py',
'my_complex.py', 'random_remarks.py', 'reverse.py']
```

10.3 Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in `sys` module's `argv` attribute as a list. For instance the following output results from running `python demo.py one two three` at the command line:

```
duangdeiMac:9 duang$ python -i bag.py one two three
>>> import sys
>>> print sys.argv
['bag.py', 'one', 'two', 'three']
```

The `getopt` module processes `sys.argv` using the conventions of the Unix `getopt()` function. More powerful and flexible command line processing is provided by the `argparse` module.

10.4 Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

10.5 String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expression offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the hat')
'cat in the hat'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Mathematics

The `math` module gives access to the underlying C library functions for floating point math:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.7071067811865476
>>> math.log(1024, 2)
10.0
```

The `random` module provides tools for making random selections:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)
[87, 8, 32, 89, 80, 42, 74, 66, 70, 78]
>>> random.random()
0.629621838712391
>>> random.randrange(6)
5
```

10.7 Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib2` for retrieving data from URLs and `smtplib` for sending mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line: # look for Eastern Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Note that the first website is not accessible for me, and the second example needs a mailserver running on localhost.)

So I wrote an example by myself, to get tomorrow's weather in Qingdao:

```
import urllib2

count = 0
flag = False

for line in urllib2.urlopen('http://www.weather.com.cn/weather/101120201.shtml'):
    if flag:
        count = count + 1

    if 'Tomorrow' in line:
        flag = True

    if count == 3:
        print line
```

Note: I didn't process the output string, so it will display some HTML labels like:

```
<p title="Sunny" class="wea">Sunny</p>
```

10.8 Dates and Times

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member

extraction member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2017, 9, 11)
>>> now.strftime("%d-%d-%y. %d %b %& is a %A on the %d day of %B.")
'11-11-17. 11 Sep & is a Monday on the 11 day of September.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
19400
```

10.9 Data Compression

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `zipfile` and `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)    # compute a CRC-32 checksum
226805979
```

10.10 Performance Measurement

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The `timeit` module quickly demonstrates a modest performance advantage:

```
>>> Timer('t = a; a = b; b = t', 'a = 1; b = 2').timeit()
0.027740955352783203
>>> Timer('a, b = b, a', 'a = 1; b = 2').timeit()
0.024269819259643555
```

In contrast to `timeit`'s fine level of granularity, the `profile` and `pstats` modules provide tools for identifying time critical sections in larger blocks of code.

10.11 Quality Control

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The `doctest` module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the `doctest` module to make sure the code remains true to documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()    # automatically validate the embedded tests
```

Use `python average.py` in terminal to run script `average.py`. If the function `average` can get the same output as written in the docstring, nothing will happen; but if I change the clause `return sum(values, 0.0) / len(values)` to `return sum(values, 1.0) / len(values)`, then `python average.py` again in terminal, we will get the following message:

```
duangdeiMac:10 duang$ python average.py
*****
File "average.py", line 4, in __main__.average
Failed example:
    print average([20, 30, 70])
Expected:
    40.0
Got:
    40.3333333333
*****
1 items had failures:
  1 of   1 in __main__.average
***Test Failed*** 1 failures.
```

The `unittest` module is not as effortless as the `doctest` module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()    # calling from the command line invokes all tests
```

We can execute `python average.py average_test.py` to test that.

10.12 Batteries Included

Python has a “batteries included” philosophy. (Motto of the Python programming language, meaning it comes with a large library of useful modules.) This is best seen through the sophisticated and robust capabilities of its larger package. For example:

- The `xmlrpcclient` and `SimpleXMLRPCServer` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.

- The `email` package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike `smtpplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachment) and for implementing internet encoding and header protocols.
- The `xml.dom` and `xml.sax` packages provide robust support for parsing this popular data interchange format. Likewise, the `csv` module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.

11 Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1 Output Formatting

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the “pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]
>>> t
[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta', 'yellow'], 'blue']]
>>> pprint.pprint(t, width = 30)
[[['black', 'cyan'],
  'white',
  ['green', 'red']],
 [['magenta', 'yellow'],
  'blue']]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns a list of
... strings instead of one big string with newlines to separate the wrapped lines."""
>>> print textwrap.fill(doc, width = 40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
>>> print textwrap.fill(doc, width = 60)
The wrap() method is just like fill() except that it returns
a list of strings instead of one big string with newlines to
separate the wrapped lines.
```

The `locale` module accesses a database of culture specific data formats. The grouping attribute of `locale`’s `format` function provides a direct way of formatting numbers with group separators:


```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()           # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                               conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

(Note: I didn't get this output on my computer.)

11.2 Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$$` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village = 'Nottingham', cause = 'the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the `safe_substitute()` method may be more appropriate — it will leave placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item = 'unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/string.py",
    line 176, in substitute return self.pattern.sub(convert, self.template)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/string.py",
    line 166, in convert val = mapping[named]
KeyError: 'owner'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d = date, n = i, f = ext)
...     print '{0} --> {1}'.format(filename, newname)
```

```
...
img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3 Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes "H" and "I" represent two and four byte unsigned numbers respectively. The "<" indicates that they are standard size and in little-endian byte order:

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):                                # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start : start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start : start+filenamesize]
    start += filenamesize
    extra = data[start : start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size                # skip to the next header
```

(Note: I failed to test this example with a customized ZIP file.)

11.4 Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
```

```
background.start()
print 'The main program continues to run in foreground.'

background.join()    # Wait for the background task to finish
print 'Main program waited until background was done.'
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the Queue module to feed that thread with requests from other threads. Applications using Queue.Queue objects for interthread communication and coordination are easier to design, more readable, and more reliable.

11.5 Logging

The logging module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```
>>> import logging
>>> logging.debug('Debugging information')
>>> logging.info('Informational message')
>>> logging.warning('Warning: config file %s not found', 'server.conf')
WARNING:root:Warning: config file server.conf not found
>>> logging.error('Error occurred')
ERROR:root:Error occurred
>>> logging.critical('Critical error -- shutting down')
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

11.6 Weak References

Python does automatic memory management (reference counting for most objects and garbage collection to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent. The `weakref` module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a          # does not create a reference
>>> d['primary']              # fetch the object if it is still alive
10
```

```

>>> del a                # remove the one reference
>>> gc.collect()         # run garbage collection right away
0
>>> d['primary']          # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/weakref.py",
    line 70, in __getitem__
      o = self.data[key]()
KeyError: 'primary'

```

11.7 Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array()` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two bytes unsigned binary numbers (typecode "H") rather than the usual 16 bytes per entry for regular lists of Python int objects.

```

>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])

```

The `collections` module provides a `deque()` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first search algorithms:

```

>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1
>>> d
deque(['task2', 'task3', 'task4'])

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
    unsearched.append(m)

```

In addition to alternative list implementations, the library also offers other tools such as the `bisect` module with functions for manipulating sorted lists:

```

>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]

```

The `heapq` module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```

>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                # rearrange the list into heap order
>>> data
[0, 1, 2, 6, 3, 5, 4, 7, 8, 9]
>>> heappush(data, -5)           # add a new entry
>>> data
[-5, 0, 2, 6, 1, 5, 4, 7, 8, 9, 3]
>>> [heappop(data) for i in range(3)] # fetch a three smallest entries
[-5, 0, 1]

```

11.8 Decimal Floating Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- financial applications and other uses which require exact decimal representation
- control over precision
- control over rounding to meet legal or regulatory requirements
- tracking of significant decimal places
- applications where the user expects the results to match calculations done by hand

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```

>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01'))
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73

```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representations enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```

>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False

```

The `decimal` module provides arithmetic with as much precision as needed:

```

>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')

```

12 What Now?

Reading this tutorial has probably reinforced your interest in using Python — you should be eager to apply Python to solving your real-world problems. Where should you go to learn more?

This tutorial is part of Python’s documentation set. Some other documents in the set are:

- The Python Standard Library
You should browse through this manual, which gives complete (though terse) reference material about types, functions, and the modules in the standard library. The standard Python distribution includes a *lot of* additional code. There are modules to read Unix mailboxes, retrieve documents via HTTP, generate random numbers, parse command-line options, write CGI programs, compress data, and many other tasks. Skimming through the Library Reference will give you an idea of what’s available.
- Installing Python Modules (Legacy version) explains how to install external modules written by other Python users.
- The Python Language Reference: A detailed explanation of Python’s syntax and semantics. It’s heavy reading, but is useful as a complete guide to the language itself.

More Python resources:

- <https://www.python.org/>: The major Python Web site. It contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your geographical location.
- <https://docs.python.org/>: Fast access to Python’s documentation.
- <https://pypi.python.org/pypi>: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- <https://code.activestate.com/recipes/langs/python/>: The Python Cookbook is a sizable collection of code examples, larger modules, and useful scripts. Particularly notable contributions are collected in a book also titled Python Cookbook (O’Reilly & Associates, ISBN 0-596-00797-3.)

13 Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the GNU Readline library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won’t duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the Unix and Cygwin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond’s PythonWin package or the Tk-based environment, IDIE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

13.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: **Ctrl-A** moves the cursor to the beginning of the line, **Ctrl-B** moves it one position to the left, **Ctrl-F** to the right. Backspace erases the character to the left of the cursor, **Ctrl-D** the character to its right. **Ctrl-K** kills (erases) the rest of the line to the right of the cursor, **Ctrl-Y** yanks back the last killed string. **Ctrl-underscore** undoes the last change you made; it can be repeated for cumulative effect.

13.2 History Substitution

History substitution works as follows. All non-empty input lines are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. **Ctrl-P** moves one line up (back) in the history buffer, **Ctrl-N** moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the **Return** key passes the current line to the interpreter. **Ctrl-R** starts an incremental reverse search; **Ctrl-S** starts a forward search.

13.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called `~/.inputrc`. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for **Tab** in Python is to insert a **Tab** character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your `~/.inputrc`. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using **Tab** for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the **Tab** key to the completion function, so hitting the **Tab** key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as `os`, which turn out to be needed in most sessions with the interpreter.

```

# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it: "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath

```

13.4 Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is IPython, which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is bpython.

14 Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction 0.125 has value $1/10 + 2/100 + 5/1000$, and in the same way the binary fraction 0.001 has value $0/2 + 0/4 + 1/8$. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximately by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at the first in base 10. Consider the fraction $1/3$. You can approximate that as a base 10 fraction: 0.3, or, better, 0.33, or, better, 0.333, and so on. No matter how many digits you're willing to write down, the result will never be exactly $1/3$, but will be an increasingly better approximation of $1/3$.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, $1/10$ is the infinitely repeating fraction

0.000110011001100110011001100110011001100110011001100110011...

Stop at any finite number of bits, and you get an approximation.

On a typical machine running Python, there are 53 bits of precision available for a Python float, so the value stored internally when you enter the decimal number 0.1 is the binary fraction


```
0.000110011001100110011001100110011001100110011001100110011010
```

which is close to, but not exactly equal to, $1/10$.

It's easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead

```
>>> 0.1
0.1
```

It's important to realize that this is, in real sense, an illusion: the value in the machine is not exactly $1/10$, you're simply rounding the *display* of the true machine value. This fact becomes apparent as soon as you try to do arithmetic with these values:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Note that this is the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Other surprises follow from this one. For example, if you try to round the value 2.675 to two decimal places, you get this:

```
>>> round(2.675, 2)
2.67
```

The documentation for built-in `round` function says that it rounds to the nearest value, rounding ties away from zero. Since the decimal fraction 2.675 is exactly halfway between 2.67 and 2.68, you might expect the result here to be (a binary approximation to) 2.68. It's not, because when the decimal string 2.675 is converted to a binary floating-point number, it's again replaced with a binary approximation, whose exact value is

```
2.67499999999999982236431605997495353221893310546875
```

Since this approximation is slightly closer to 2.67 than to 2.68, it's rounded down.

If you're in a situation where you care which way your decimal halfway-cases are rounded, you should consider using the `decimal` module. Incidentally, the `decimal` module also provides a nice way to “see” the exact value that's stored in any particular Python float

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
```

Another consequence is that since 0.1 is not exactly $1/10$, summing ten values of 0.1 may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

Binary floating-point arithmetic holds many surprises like this. The problem with “0.1” is expected in precise detail below, in the “Representation Error” section. See *The Perils of Floating point* for a more complete account of other common surprises.

At that says near the end, “there are no easy answers.” Still, don’t be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2^{53} per operation. That’s more than adequate for most tasks, but you do need to keep in mind that it’s not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you’ll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. For line control over how a float is displayed see the `str.format()` method’s format specifiers in [Format String Syntax](#).

14.1 Representation Error

This section explains the “0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won’t display the exact decimal number you expect:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Why is that? $1/10$ and $2/10$ are not exactly representable as a binary fraction. Almost all machines today (July 2010) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 “double precision”. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form $J/2^N$ where J is an integer containing exactly 53 bits. Rewriting

$$1 / 10 \approx J / (2^N)$$

as

$$J \approx 2^N / 10$$

and recalling that J has exactly 53 bits (is $\geq 2^{52}$ but $< 2^{53}$), the best value for N is 56:

```
>>> 2**52
4503599627370496
>>> 2**53
9007199254740992
>>> 2**56/10
7205759403792794.0
```

That is, 56 is only value for N that leaves J with exactly 53 bits. The best possible value for J is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> q
7205759403792793
>>> r
6
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to $1/10$ in 754 double precision is that over 2^{56} , or

```
>>> 7205759403792794 / 72057594037927936
0.1
```

Note that since we rounded up, this is actually a little bit larger than $1/10$; if we had not rounded up, the quotient would have been a little bit smaller than $1/10$. But in no case can it be *exactly* $1/10$!

So the computer never “sees” $1/10$: what it sees is the exact fraction given above, the best 754 double approximation it can get:

```
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10^{**30} , we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794 * 10**30 // 2**56
1000000000000000005551115123125
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 1000000000000000005551115123125. In version prior to Python 2.7 and Python 3.1, Python rounded this value to 17 significant digits, giving ‘0.10000000000000001’. In current versions, Python displays a value based on the shortest decimal fraction that rounds correctly back to the true binary value, resulting simply in ‘0.1’.